Python
Very Easy and Flexible set of Built-In Containers.

I- Containers and Data Structures

Counting frequencies   count occurrences of any hashable value

Dictionary with fallback—have a fallback value for any missing key

Unpacking multiple—keyword arguments—how to use ** more than once

Ordered dictionaries—maintaining order of keys in a dictionary

MultiDict—dictionary with multiple values per key

Prioritizing entries—efficiently get the top of sorted entries

Bunch—dictionaries that behave like objects

Enumerations—handle a known set of states

II-Text Management

- ❑Pattern matching—regular expressions are not the only way to parse patterns; Python provides easier and just as powerful tools to parse patterns
- ❑Text similarity—detecting how two similar strings in a performing way can be hard but Python has some easy-to-use built-in tools
- ❑Text suggestion—Python looks for the most similar one to suggest to the user the right spelling
- ❑Templating—when generating text, templating is the easiest way to define the rules
- ❑Splitting strings preserving spaces—splitting on empty spaces can be easy, but gets harder when you want to preserve some spaces
- ❑Cleanup text—removes any punctuation or odd character from text
- ❑Normalizing text—when working with international text, it's often convenient to avoid having to cope with special characters and misspelling of words
- ❑Aligning text—when outputting text, properly aligning it greatly increases readability

III-Command Line

Basic logging—logging allows you to keep track of what the software is doing, and it's usually unrelated to its output

Logging to file—when logging is frequent, it is necessary to store the logs on a disk

Logging to Syslog—if your system has a Syslog daemon, you might want to log in to Syslog instead of using a standalone file

Parsing arguments—when writing with command-line tools, you need parsing options for practically any tool

Interactive shells—sometimes options are not enough and you need a form of Read-Eval-Print Loop to drive your tool

Sizing terminal text—to align the displayed output properly, we need to know the terminal window size

Running system commands—how to integrate other third-party commands in your software

Why Command Line ?

- ❑When writing a new tool, one of the first needs is ble to interact with the surrounding environment
  - ❖to display results, track errors, and receive inputs.
- ❑Users are accustomed to certain standard ways a command-line tool interacts with them and with the system
- ❑Standard library in Python provides tools to achieve the most common needs in implementing software that is able to interact through a shell and through text.

Some of them are:

- ❖to implement some forms of logging, so that the program can keep a log file
- ❖to implement both options-based and interactive software,
- ❖to implement more advanced graphical output based on text.

IV-Filesystem and Directories

- Traversing folders—recursively traversing a path in the filesystem and inspecting its contents
- Working with paths—building paths in a system-independent way
- Expanding filenames—finding all files that match a specific pattern
- Getting file information—detecting the properties of a file or directory
- Named temporary files—working with temporary files that you need to access from other processes too
- Memory and disk buffer—spooling a temporary buffer to disk if it's bigger than a threshold
- Managing filename encoding—working with the encoding of filenames
- Copying a directory—copying the content of a whole directory
- Safely replacing a file's content—how to replace the content of a file safely in case of failures

IV-Filesystem and Directories

- As a developer, working with files and directories can be more complex than expected
  - when multiple platforms have to be supported or encodings are involved.
- The Python standard library has many powerful tools to work with files and directories.
- It's clear that the standard library provides a great set of tools to work with files and directories.
  - os, shutil, stat, and glob functions have been defined

V- Date and Time

- ❑Time-zone-aware datetime—retrieving a reliable value for the current datetime
- ❑Parsing dates—how to parse dates according to the ISO 8601 format
- ❑Saving dates—how to store datetimes
- ❑From timestamp to datetime—converting to and from timestamps
- ❑Displaying dates in a user format—formatting dates according to our user language
- ❑Going to tomorrow—how to compute a datetime that refers to tomorrow
- ❑Going to next month—how to compute a datetime that refers to next month
- ❑Weekdays—how to build a date that refers to the nth Monday/Friday of the month
- ❑Workdays—how to get workdays in a time range
- ❑Combining dates and times—making a datetime out of a date and time

VI- Read/Write Data

- Reading and writing text data—reading text encoded in any encoding from a file
- Reading lines of text—reading a text file divided line by line
- Reading and writing binary data—reading binary-structured data from a file
- Zipping a directory—reading and writing a compressed ZIP archive
- Pickling and shelving—how to save Python objects on disk
- Reading configuration files—how to read configuration files in the .ini format
- Writing XML/HTML content—generating XML/HTML content
- Reading XML/HTML content—parsing XML/HTML content from a file or string
- Reading and writing CSV—reading and writing CSV spreadsheet-like files
- Reading and writing to a relational database—loading and saving data into a SQLite database

Algorithms

- Searching, sorting, filtering—high-performance searching in sorted containers
- Getting the nth element of any iterable—grabbing the nth element of any iterable
- Grouping similar items—splitting an iterable into groups of similar items
- Zipping—merging together data from multiple iterables into a single iterable
- Flattening a list of lists—converting a list of lists into a flat list
- Producing permutations and—computing all possible permutations of a set of elements
- Accumulating and reducing—applying binary functions to iterables
- Memoizing—speeding up computation by caching functions
- Operators to functions—how to keep references for a Python operator
- Partials—reducing the number of arguments of a function
- Generic functions—functions that are able to change behavior according to the provided argument type
- Proper decoration—properly decorating a function to avoid missing its signature and docstring
- Context managers—automatically running code whenever you enter and exit a block of code

Containers and Data Structure

1-Counting Frequencies

- A very common need in many kinds of programs is to count the occurrences of a value or of an event
  - This means counting frequency.
    - ✓ count words in text
    - ✓ count likes on a blog post
    - ✓ track scores for players of a video game
- In the end counting frequency means counting how many we have of a specific value

1-Counting Frequencies

- The most obvious solution for such a need is to count.
- If there are two, three, or four, maybe we can just track them
- if there are hundreds, it's certainly not feasible to keep around such a large amount of variables
- we will quickly end up with a solution based on a container to collect all those counters.

How to Count Frequencies

❑To track the frequency of words in text, the standard library comes and provides to track counts and frequencies

collections. Counter object.

i) keeps track of frequencies,

iii) provides some dedicated methods to retrieve the most common entries,

iii) entries that appear at last once and quickly count any iterable.

Any iterable priveded to the **Counter** is "counted" for its frequency of values:

```
>>> txt = "This is a vast world you can't traverse world in a day"
>>>
>>> from collections import Counter
>>> counts = Counter(txt.split())
```

- The result would be a dictionary with the frequencies of the words in the phrase:

```
Counter({'a': 2, 'world': 2, "can't": 1, 'day': 1, 'traverse': 1,         'is': 1,
'vast': 1, 'in': 1, 'you': 1, 'This': 1})
```

Operations on Counters

- query for the most frequent words:

```
>>> counts.most_common(2)
[('world', 2), ('a', 2)]
```

- Get the frequency of a specific word:

```
>>> counts['world']
     2
```

- get back the total number of occurrences

```
>>> sum(counts.values())
     12
```

Some set Operations on Counters

- ❑Joining counters

```
>>> Counter(["hello", "world"]) + Counter(["hello", "you"])
Counter({'hello': 2, 'you': 1, 'world': 1})
```

- ❑Checking counters for intersections

```
>>> Counter(["hello", "world"]) & Counter(["hello", "you"])
Counter({'hello': 1})
```

How Counter Works?

- counting code relies on the fact that Counter is just a special kind of dictionary,
- Dictionaries can be built by providing an iterable.
- Each entry in the iterable will be added to the dictionary
- In the case of a counter, adding an element means incrementing its count; for every "word" in our list, we add that word multiple times (one every time it appears in the list)
  - ✓ so its value in the Counter continues to get incremented every time the word is encountered.

Counter is a dictionary

- ❑Counter is actually not the only way to track frequencies
- ❑Counter is a special kind of dictionary

Reproducing the Counter behavior:

```
        counts = dict(hello=0, world=0, nice=0, day=0)
```

For a new occurrence of hello, world, nice, or day, it is required to increase the associated value in the dictionary

```
   for word in 'hello world this is a very nice day'.split():
    if word in counts:       counts[word] += 1
```

By relying on *dict.get*, we can also easily adapt it to count any word

```
for word in 'hello world this is a very nice day'.split():
 counts[word] = counts.get(word, 0) + 1
```

- The standard library actually provides a very flexible tool that we can use to improve the code   collections.defaultdict.
- defaultdict(int) will create a dictionary that provides 0 for any key that it doesn't have.

```
 from collections import defaultdict
 counts = defaultdict(int)
for word in 'hello world this is a very nice day'.split():
  counts[word] += 1
```

The result will be exactly what we expect

```
defaultdict(<class 'int'>, {'day': 1, 'is': 1, 'a': 1, 'very': 1, 'world': 1, 'this':
1, 'nice': 1, 'hello': 1})
```

for each word, the first time we face it, we will call int to get the starting value and then add 1 to it.

As int gives 0 when called without any argument

The Most Frequent Entry in Bag of Words

- ❑The convenience of Counter is based on the set of additional features
- ❑ Counter is not just a dictionary with a default numeric value,
- ❑Counter is a class specialized in keeping track of frequencies
- ❑Counter is a class providing convenient ways to accessbag of words

Dictionary with fallback

- ❑When working with configuration values, it's common to look them up in multiple places—
  - ❖We can load them from a configuration file—
  - ❖we can override them with an environment variable or a command-line option,
  - ❖ and in case the option is not provided, we can have a default value.
- ❑This can easily lead to long chains of if statements like these:

```
value = command_line_options.get('optname‘)
if value is None:
     value = os.environ.get('optname')
if value is None:
     value = config_file_options.get('optname')
if value is None:
    value = 'default-value'
```

Dictionary with fallback

- Command-line options are a very frequent use case
  - but the problem is related to chained scopes resolution.
- Variables in Python are resolved by looking at locals();
  - if they are not found, the interpreter looks at globals()
  - if they are not yet found, it looks for **built-ins**.
- The alternative for chaining default values of dict.get   instead of using multiple if instances
- If we want to add one additional scope, we would have to add it in every single place where we are looking up the values.collections.ChainMap
- We can provide a list of mapping containers and it will look for a key through them all.

Dictionary with fallback

- ❑The previous example involving multiple different if instances can be converted to something like this:

```
import os
from collections import  ChainMap
options = ChainMap(command_line_options,  os.environ,
                                                  config_file_options)
value = options.get('optname', 'default-value')
```

Dictionary with fallback

- We can also get rid of the last .get call by combining ChainMap with defaultdict.
- we can use defaultdict to provide a default value for every key

```
import os
from collections import ChainMap, defaultdict
options = ChainMap(command_line_options,    os.environ,
                                   config_file_options,   defaultdict(lambda: 'default-value'))
value = options['optname']
value2 = options['other-option']
```

Dictionary with fallback

Print value and value2 will result in the following:

```
Optvalue
 default-value
```

optname will be retrieved from the command_line_options containing it, while other-option will end up being resolved by defaultdict.

How ChainMap Works?

- The ChainMap class receives multiple dictionaries as arguments
- Whenever a key is requested to ChainMap, it goes through the provided dictionaries one by one to check whether the key is available in any of them.
- Once the key is found, it is returned, as if it was a key owned by ChainMap itself.

How ChainMap Works?

- ❑The default value for options that are not provided is implemented by having defaultdict as the last dictionary provided to ChainMap.
- ❑Whenever a key is not found in any of the previous dictionaries, it gets looked up in defaultdict
  - ❖ which uses the provided factory function to return a default value for all keys.

Other ChainMap Features

- ❑Another feature of ChainMap is to allow updating
  - ❖but instead of updating the dictionary where it found the key, it always updates the first dictionary.
- ❑The result is the same, as on next lookup of that key,
- ❑ we would have the first dictionary override any other value for that key (as it's the first place where the key is checked).
- ❑The advantage is that if we provide an empty dictionary as the first mapping provided to ChainMap, we can change those values without touching the original container

Example: ChainMap

```
>>> population=dict(italy=60, japan=127, uk=65)
>>> changes = dict()
>>> editablepop = ChainMap(changes, population)
>>> print(editablepop['japan'])
127
>>> editablepop['japan'] += 1
>>> print(editablepop['japan'])
128
```

Example: ChainMap

- But even though we changed the population of Japan to 128 million, the original population didn't change:

```
>>> print(population['japan'])
127
```

- we can use changes to find out which values were changed and which values were not:

```
>>> print(changes.keys())
dict_keys(['japan'])
 >>> print(population.keys() - changes.keys())
{'italy', 'uk'}
```

Example: ChainMap

- if the object contained in the dictionary is mutable and we directly mutate it, ChainMap can do to avoid mutating the original object.
- Instead of numbers, we store lists in the dictionaries,
- we can mutate the original dictionary whenever we append values to the dictionary

Example: ChainMap

```
>>> citizens = dict(torino=['Alessandro'], amsterdam=['Bert'],
raleigh=['Joseph'])
 >>> changes = dict()
>>> editablecits = ChainMap(changes, citizens)
>>> editablecits['torino'].append('Simone')
 >>> print(editablecits['torino'])
 ['Alessandro', 'Simone']
>>> print(changes)
{}
>>> print(citizens)
{'amsterdam': ['Bert'],  'torino': ['Alessandro', 'Simone'],  'raleigh': ['Joseph']}
```

Unpacking Multiple eyword Arguments

- Python functions accept arguments from a dictionary through unpacking (the ** syntax)
- Given a function, f, we want to pass the arguments from two dictionaries, d1 and d2

```
>>> def f(a, b, c, d) :
...  print (a, b, c, d)
...
>>> d1 = dict(a=5, b=6)
>>> d2 = dict(b=7, c=8, d=9)
```

collections.ChainMap can help to achieve what we want; it can cope with duplicated entries

```
>>> f(**ChainMap(d1, d2))
    5 6 8 9
```

Unpacking Multiple Keyword Arguments

In Python 3.5 and newer versions, you can also create a new dictionary by combining multiple dictionaries through the literal syntax,

then pass the resulting dictionary as the argument of the function:

```
>>> f(**{**d1, **d2})
5 7 8 9
```

- ❑The duplicated entries are accepted too, but are handled in reverse order of priority to ChainMap (so right to left).
- ❑b has a value of 7, instead of the 6 it had with ChainMap, due to the reversed order of priorities.

Unpacking Multiple Keyword Arguments

ChainMap looks up keys in all the provided dictionaries

- ❖it's like the sum of all the dictionaries.

❑The unpacking operator (**) works by inviting all keys to the container and then providing an argument for each key.

❑As ChainMap has keys resulting from the sum of all the provided dictionaries keys, it will provide the keys contained in all the dictionaries to the unpacking operator

- ❖ChainMap allows us to provide keyword arguments from multiple dictionaries.